Getting to Know ForEach and ForEach-Object

Dr Scripto

July 8th, 2014

Summary: Learn the differences between **ForEach** and **ForEach-Object** in Windows PowerShell.

Honorary Scripting Guy and Windows PowerShell MVP, Boe Prox, here today filling in for my good friend, The Scripting Guy. Today I am going to talk about some differences between using **ForEach** and using **ForEach-Object** in day-to-day scripting activities.

There are times when you are unable to make use of a cmdlet that has built-in pipeline support, such as something like this:

```
Get-ChildItem –File –Filter "*.TMP" | Remove-Item –Verbose
```

To get around this, we can make use of some other capabilities of Windows PowerShell by using **ForEach** or **ForEach-Object** to iterate through collections and to perform an action against each item in the collection. Each of these approaches can let you run through a collection and then perform actions in a script block. What you may not know is that each cmdlet has two approaches to how they take and handle the collections.

Let's take a look at **ForEach-Object** and see what it is about. This cmdlet has a couple of aliases that may seem familiar to you:

```
Get-Alias –Definition ForEach-Object
```

```
CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Alias           % -> ForEach-Object
Alias           foreach -> ForEach-Object
```

Wait a second! Why in the world are there two **ForEach** options in Windows PowerShell? This is an excellent question, and fortunately, I have an answer. When you are piping input into **ForEach**, it is the alias for **ForEach-Object**. But when you place **ForEach** at the beginning of the line, it is a Windows PowerShell statement.

ForEach-Object is best used when sending data through the pipeline because it will continue streaming the objects to the next command in the pipeline, for example:

```
ForEach-Object -InputObject (1..1E4) {
  $_
} | Measure-Object


Count    : 10000
Average  :
Sum      :
Maximum  :
```

```
Minimum  :

Property :
```

You cannot do the same thing with **ForEach () {}** because it will break the pipeline and throw error messages if you attempt to send that output to another command.

```
ForEach ($i in (1..1E4)) {
   $i
} | Measure-Object


At line:3 char:3
+ } | Measure-Object
+   ~
```

Note that an empty pipe element is not allowed.

```
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : EmptyPipeElement
```

You would have to save all of the output that is being process by **ForEach** to a variable and then pipe it to another cmdlet, for example:

```
$Data = ForEach ($i in (1..1E4)) {
   $i
}
$Data | Measure-Object
```

The fact that now we have totally broken the pipeline becomes more apparent after this. Not only do we have to stop the pipeline to begin processing the data, we cannot even send that data to the pipeline from the statement without first collecting the output into a variable and then sending it down the pipeline.

This is very important if you plan to use the data in another command through the pipeline. It is important to note another difference that these options share, which is performance vs. memory consumption.

The **ForEach** statement loads all of the items up front into a collection before processing them one at a time. **ForEach-Object** expects the items to be streamed via the pipeline, thus lowering the memory requirements, but at the same time, taking a performance hit. Following are a couple of tests to highlight the differences between these:

```
$time = (Measure-Command {
   1..1E4 | ForEach-Object {
      $_
   }
}).TotalMilliseconds
 [pscustomobject]@{
   Type = 'ForEach-Object'
   Time_ms = $Time
 }
```

```
$Time = (Measure-Command {
    ForEach ($i in (1..1E4)) {
        $i
    }
}).TotalMilliseconds
 [pscustomobject]@{
    Type = 'ForEach_Statement'
    Time_ms = $Time
 }
```

```
Type                                                                Time_ms
----                                                                -------
ForEach-Object                                                      834.855
ForEach_Statement                                                   68.7594
```

As expected, the **ForEach** statement, which allocates everything to memory before processing, is the faster of the two methods. **ForEach-Object** is much slower. Of course, the larger the amount of data, the more risk you have of running out of memory before you are able to process all of the items. So be sure to take that into consideration.

To throw another curve ball into this, check out this alternate approach to **ForEach-Object**. This time, we'll use the **InputObject** parameter (this is the parameter used in the pipeline process):

```
$Time = (Measure-Command {
    ForEach-Object -InputObject (1..1E4) {
        $_
    }
}).TotalMilliseconds
 [pscustomobject]@{
    Type = 'ForEach-Object_Param'
    Time_ms = $Time
 }
```

```
Type                                                                Time_ms
----                                                                -------
ForEach-Object_Param                                                 4.9069
```

Wow, that was fast! Why am I not talking this up instead of focusing on **ForEach**? Although this seems like the fastest of the three approaches, the major (yes, major!) issue is that we are being deceived into thinking that it just processed everything in an amazing amount of time. But the fact is that all we did was pass the entire collection to the script block one time—and that was it.

```
ForEach-Object -InputObject (1..1E4) {
    $_.GetType().FullName
}
```

```
PS C:\> ForEach-Object -InputObject (1..1E4) {
    $_.GetType().FullName
}
System.Object[]
```

Well, it was worth a shot to squeeze a little more speed out of this. But in the end, we get something that is completely unusable—even if we did want to send it down the pipeline.

ForEach-Object also allows us to specify **Begin**, **Process**, and **End** script blocks that we can use (similar to an advanced function) to set up our environment, process each item, and then do something (such as clean up at the end of the command).

```
Get-ChildItem -Force | ForEach-Object -Begin {
   Write-Verbose "Begin block" -Verbose
} -Process {
   If ($_.length -gt 555) {
      Write-Verbose "Process block" -Verbose
      $_
   }
} -End {
   Write-Verbose "End block" -Verbose
}
```

```
VERBOSE: Begin block
VERBOSE: Process block

    Directory: C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-arhs         3/18/2014   5:02 AM     398356 bootmgr
VERBOSE: Process block
-arhs         6/18/2014  12:37 AM       8192 BOOTSECT.BAK
VERBOSE: Process block
-a-hs         6/25/2014  10:39 PM 1342177280 pagefile.sys
VERBOSE: End block
```

Here you see that the **Begin** block kicks off first, followed by all of the items that I am processing and filtering, with the **End** block being processed last. If I wanted, I could then pass this to another cmdlet, such as **Export-CSV**. You couldn't come close to doing this type of action by using the **ForEach** statement.

So which one do you use? Well, the answer is, "It depends."

You can iterate through a collection of items by using either the **ForEach** statement or the **ForEach-Object** cmdlet.

- **ForEach** is perfect if you have plenty of memory, want the best performance, and do not care about passing the output to another command via the pipeline.
- **ForEach-Object** (with its aliases **%** and **ForEach**) take input from the pipeline. Although it is slower to process everything, it gives you the benefit of **Begin**, **Process**, and **End** blocks. In addition, it allows you to stream the objects to another command via the pipeline.

In the end, use the approach that best fits your requirement and the capability of your system.

Follow the Scripting Guys on Twitter and Facebook. If you have any questions, send an email to the Scripting Guys at scripter@microsoft.com, or post your questions on the Official Scripting Guys Forum.

Boe Prox, Windows PowerShell MVP and Honorary Scripting Guy

Dr Scripto

Scripter, PowerShell, vbScript, BAT, CMD

Follow Dr Scripto

Posted in Uncategorized Tagged Boe Prox, getting started, guest blogger, Scripting Guy!, scripting techniques, Windows PowerShell

Log in to comment

Log In

Jeremy Bradshaw 2019-05-15 05:22:22

Dear Dr. Scripto, Thanks for the great article, and hopefully I'm not too late to ask a question (just 4 years late). Would you be able to comment on when/how the additional memory is free'd back up after using the foreach statement? I've almost exclusively switched a lot of my scripts over to ForEach-Object due to high memory consumption, but I think I could benefit from the speed of foreach statement for many situations. But before I do, I'd really like to know if/when the memory that is consumed by foreach statement will be free for use again. If it isn't until after my script is finished executing, then I guess this means I'd have to use garbage collection in my script to keep on top of that, and I think I've read a few posts saying to be careful with garbage collection and leave it to PowerShell. Would love to get your feedback on this particular subtopic. Thanks, Jeremy

Relevant Links

Scripting Forums

PowerShell Forums

PowerShell on TechCommunity

PowerShell.org – Community owned resources for PowerShell

Top Bloggers

Doctor Scripto
Scripter

ScriptingGuy1

mredwilson

CraigLieb

I_am_mr_ed

Archive

Stay informed